On Implementing Separate Compilation

In Block-Structured Languages

Richard J. LeBlanc
School of Information and Computer Science
Georgia Institute of Technology

Charles N. Fischer
Computer Sciences Department
University of Wisconsin - Madison

Introduction

Perhaps the single most important paradigm of modern programming language design is block structure. Block-structured languages are characterized by nested definitional units (ALGOL blocks, PASCAL procedures, Euclid modules) having rather specialized scoping rules. In particular, access to entities defined in containing units is allowed (although sometimes with restrictions) while access to entities defined within a unit from outside it is severely restricted or totally forbidden.

This method of programming language structuring supports a top-down program development methodology: the body of a definition unit can be developed (or modified) without affecting other units (since its internal details are "hidden" from the outside). Such units (most notably procedures) become the natural units of program development and modification.

However, program development and maintenance in block-structured languages can have some drawbacks. Conceptually, programs in such languages are monolithic -- the entire text of a program is needed to do a compilation. This complicates program development and modification as even a single line change can (and will) force a recompilation of the entire program.

As a result, separate compilation facilities are invariably added to production compilers. Such facilities allow a program to be composed of individual units (usually the definitional units of a language) which can be separately compiled. These units are then collected together (via a linkage editor) into an executable program.

This approach is essentially the FORTRAN external subroutine model and, when applied to block-structured languages, suffers some rather severe drawbacks. Individual units under such separate compilation models are very limited in how they can share information. They are, in fact, commonly limited to sharing entry point names and simple shared variables (e.g., COMMON blocks). Further, since compilations are truly separate, a compiler has no opportunity to verify that interfaces between units (e.g., procedure calls) are correct. Any checking that is done must occur at run-time or during linkage of separate segments, and even then complete checking is difficult to obtain (since, for example, type rules vary among languages).

The PASCAL 6000 compiler [1] for Control Data computers does include an external procedure facility with no checking across interfaces at all. This is certainly contrary to the design philosophy of PASCAL. The NU ALGOL compiler [2] for UNIVAC 1100 series machines implements external procedures with run-time checking. It generates data structures which represent the declaration of the procedure and the external declaration in the block from which it is called. These are compared when the procedure is executed to insure compatibility. A PASCAL compiler for IBM 360 computers [3] has been developed which generates data structures to be utilized by a linkage program which performs inter-module checking while creating an object module from the various segments presented to it. While these last two techniques do detect errors, the detection is delayed to some time after compilation is completed. In one case run-time inefficiency is introduced and in the other an extra step is added to the process of preparing a program for execution, requiring the development and maintenance of an additional piece of software.

A simple separate compilation scheme

The question then naturally arises as to whether separate compilation techniques must weaken the definition sharing and compile-time checking normally found in block-structured languages. Recent work involving the development of separate compilation facilities in the UW-PASCAL compiler [4,5] shows that the answer is no -- separate compilation can be viewed as a compiler implementation issue which need not impact the fundamental design of a programming language or its compiler.

Because of the scoping rules imposed by block-structured languages, their compilers normally translate a definitional unit (e.g., a procedure) in two steps. First, all definitions in containing units which may be referenced by the definitional unit are processed. Then, using these definitions (call them an environment), the unit itself is translated. Now if this environment can be saved, compilation of the unit can be deferred or various versions of the unit can be compiled using the same environment. In effect, the compiler is checkpointed, and the environment represents the checkpoint information needed to restart the compiler. This approach, although conceptually simple, is remarkably powerful. No changes at all are needed in the the programming language or its definition as conceptually all programs are still monolithic. Further, the compiler itself can be designed as if all compilations are monolithic. Then only the slightest changes are needed to allow the environment of a unit to be saved and later reinstated.

Under this approach full sharing of all accessible definitions is guaranteed as is the compile-time checking of all interfaces between individual units (even if they are separately compiled). In order to compile units separately, a program's text is divided into segments. The first segment to be compiled is a main segment. This segment contains the declarations of all constants, types, variables and other entities which may be defined (and shared) at a global level. It also contains the bodies of all definitional units which are to be compiled immediately as well as "stubs" for those units which are to be separately compiled. These stubs contain the information necessary to allow compile-time checking of the use of the units. In the case of a procedure, this would require the name, the number and types of parameters, and a return value type, if any. Stubs may appear only at the global declaration level and are combined with all of the other global declarations in a main segment to define an environment. Stubs are not allowed within other definitional units so that only this single environment is necessary to compile the bodies of the units they represent.

Any entity defined in a main segment may be referenced in the normal manner in a segment compiled using the environment containing its definition. This mechanism allows definitional units compiled separately to reference each other (for example, procedures compiled separately may call one another) and to share the same global declarations, including variables, constants, types, etc. Compilation of a main segment creates a file which contains an encoding of an environment. The name of the file to be created can be specified within the text of the segment or as part of the call to the compiler. The latter approach is certainly more flexible, but depends on the available interface between the compiler and the operating system. The environment contains the information needed do all type checking during compilation of other segments, just as if all of the separately compiled units had been compiled together in a standard program.

Once an environment file has been created, it can be used to compile any number of deferred segments. Each of these can include the complete definition of one or more of the definitional units for which stubs appeared in the main segment. Again we assume some convention for specifying an environment file to be used during compilations of the units within the segment. Units which were not declared as part of the environment may also appear within a deferred segment. Since there is no information about them in the environment, they may be referenced only by other units in the segment in which they appear.

To insure consistency of usage and to enhance readability of programs, all of the information in a stub should be repeated when its text is specified in a deferred segment. The compiler can then verify that the information in the stub correctly describes the unit, and someone reading the program can see a complete definition of a unit without looking beyond the segment under consideration.

The separate compilation mechanism which has been described allows the internal details of definitional units to be hidden from other parts of a program. That is, a deferred segment may be modified and recompiled without any effect on other segments. However, this is not true for main segments -- any changes in the environment require all executable segments to be recompiled, even if they are apparently unaffected by the change (since, e.g., the address of a global variable may be changed). Two improvements are possible which minimize the number of total recompilations necessary.

The change in an environment which least affects previously compiled units is the declaration of one or more new entities in the environment. If a new kind of segment is allowed, one which has the effect of extending an existing environment, no definitional units need to be recompiled other than those which interact with these newly defined entities. These environment extension segments take the descriptor of the old environment and add to it using the new declarations, thus creating a new descriptor file. The addition of this new kind of segment allows the segments which make up a program to be compiled in a linear sequence of environments rather than only one.

Another desirable improvement is to allow variables and other entities external to any one definitional unit to be declared within a deferred segment. These definitions and variables (which are statically allocated) are available to all units in the segment. In a monolithic compilation of the program, these declarations could be made global (possibly using renaming to avoid identifier clashes), so this extension does not significantly change the semantics of the language. The benefit is in allowing changes to declarations of entities used only within one segment to be made without requiring alterations to the environment and thus possibly a recompilation of other segments. A fortunate side effect of this addition is that access to the these entities is effectively limited to only those units which need it.

Implementation considerations

A major advantage of the simplicity of these features, beyond the ease with which they may be understood by a programmer, is the ease with which they may be implemented. For example, only about two man-weeks of work were required to add separate compilation to the UW-PASCAL compiler. Beyond code to read and write environments, the only work needed was a slight extension of program syntax (to specify main and deferred segments) and the addition of external references to implement inter-segment communication. No changes in the basic program structure were necessary.

A simple, efficient technique for saving the symbol table which defines an environment can be implemented in the likely case that the symbol table can be described as a contiguous block of storage. Saving the environment requires storing any external pointers, counts or other variables which describe the state of the table in a file along with the symbol table block. These variables are perhaps not stored contiguously, but collecting their values into a block for writing to secondary storage is not a significant expense. If the size or location of the symbol table is variable, the appropriate descriptive information is included in the block with these other values. Since a single output operation is usually sufficient to write a block of information to secondary storage, given a start address and a length, an environment can be defined using only two output records. The first is the fixed size block of descriptive variables and the second is the symbol table.

To compile a deferred segment, the environment specified is restored by first loading the block of descriptive information and putting any values contained therein into the appropriate variables. Then the symbol table is reloaded, possibly using size and/or address information from the first record. The symbol table then looks just like it did after the global declarations were processed. In fact, UW-PASCAL takes an especially simple approach to saving and restoring environments. Since the compiler is itself a PASCAL program, it merely saves and restores the entire run-time heap along with a few pointers.

There are a few subtle dangers in this implementation approach when the symbol table is dynamically constructed using pointers, as is typically the case in a PASCAL compiler. One fairly simple restriction is necessary if the entire heap is to be restored to load an environment: the reload must be done before any new objects are allocated from the heap. There is the further difficulty that heap objects must be reloaded into the exact same addresses in order for pointers to them to be valid. This can cause problems when a new version of the compiler is produced. If the same address space is not available for the heap, all programs using separate compilation must be completely recompiled. These problems could be circumvented by storing some symbolic representation of the environment, but such storage would be far more difficult to implement and much less efficient than the approach which has been described.

In practice it is necessary to have some mechanism to guarantee that all of the binary files presented to a linkage editor or loader to make up a program have been compiled in the same environment. This can be accomplished by including as part of each environment a unique tag (e.g., an encoding of the date and time of creation). Each binary file, representing the compilation of a deferred segment, contains an external reference to a label generated from the tag of the environment. The binary file from

the main segment includes an external label generated from the tag. The linkage editor will then generate an error message if the reference in some segment is not satisfied, thus indicating an environment error. If extensions to an environment are allowed, a binary file is created when each of the extensions is compiled. Each such binary file contains external labels corresponding to the tags of all extended environments in the chain leading to the extension being compiled. When all of the executable modules are linked together, the binary files from the original main segment and the last extension segment are also included. Thus a label will be available to match the reference in any binary file created using any of the chain of environments.

Experience with the technique

This external compilation technique has been in use with UW-PASCAL for about 2 years. User reaction has been most favorable. Since conceptually all programs are monolithic, segmentation can be used, at any time, to reduce compilation costs or to structure the text of a large program. Nothing is lost in using segmentation and the novice user need never be burdened by its existence. The simplicity of the mechanism also helped to make its implementation cheap and (reasonably) painless.

The principal example of the utility of this separate compilation technique is the UW-PASCAL compiler itself. It was originally written to be compiled as a single unit. Then after it was enhanced to include the separate compilation capability, the source code was divided into about twenty segments and the necessary stubs were added to the main segment. Making these alterations and bringing up this separately compiled version took only a short time, but the benefits were dramatic. Because of the sheer size of the compiler and its interaction with the system workload, it had previously been practical to make changes in the compiler only by recompiling it at night. After the segmentation, it became possible to edit and recompile a segment and then rebuild the compiler all within an hour, even during very busy periods.

Comparison to other work

A similar mechanism is implemented in the ALGOL 68C compiler developed at Cambridge University [6]. The main difference between the two approaches is that the ALGOL 68C facility allows a tree of environments to be built rather than only a single global environment. While this approach gives more direct support to the methodology of top-down program development and testing, it has some disadvantages, as discussed by Schwartz [7]. The most important of these for comparison to our approach is that the environment of a particular block can only be ascertained by a programmer by reading the declarations in all of the other segments in the chain leading to the main program environment (each of these segments can contain both declarations and executable code). This is considerably more difficult for a programmer to handle than when the environment consists of only declarations within a segment and those which define the global environment. Further, the value of a tree-structured environment in practice is questionable. Experience seems to indicate that sharing of definitions occurs primarily at two levels: global definitions are shared by all and local definitions are shared within the body of a definitional unit. Our separate compilation mechanism is keyed to this usage.

A mechanism which allows the separate compilation of classes and procedures [8] has also been incorporated into several SIMULA 67 implementations. It is based on the idea of creating a descriptor file as each such object is separately compiled and then providing these descriptors to the compiler for checking purposes when other program segments which use the precompiled objects are themselves compiled. The disadvantages of this approach include the requirement of bottom-up compilation and the necessity of manipulating a potentially large number of descriptor files. Additionally, any interface between segments through shared global variables is made impossible. While it is open to question whether such sharing of variables is an advantage or disadvantage for purposes of modular program construction, it does make it impossible for programs to be written without regard for the intended mode of compilation.

The design of the language MESA [9] includes the concept of an interface, which is the basis of separate compilation capabilities for a MESA compiler. MESA programs are made up of definitions modules and program modules, with the compilation of a definitions module producing symbol table information as its output. Any number of definitions modules may be part of a program, so this is another mechanism which can create many environment files. There is a considerable amount of checking at compile-time, but an explicit, MESA specific binding step is necessary to guarantee that the interfaces between the modules used to make up a program are indeed correct.

Summary

A very simple and efficient technique for the introduction of external compilation facilities into compilers for block-structured languages has been presented. Using this technique, programs may be compiled in parts while the compile-time checking advantages of compilation as a whole are retained. These features are simple for a programmer to understand (because all programs are conceptually monolithic) and are easy to implement. This approach therefore appears to be of real value in enhancing the utility and scope of modern compilers.

Acknowledgement

Credit is due to Marty Honda, one of our co-workers on the UW-PASCAL project, for his significant contributions to the original implementation of these separate compilation techniques.

References

[1] K. Jensen and N. Wirth, PASCAL User Manual and Report, 2nd Ed., Springer-Verlag, 1977.

[2] UNIVAC 1100 Series NU ALGOL Programmer Reference Manual, Sperry Rand Corporation, 1971.

[3] R.B. Keiburtz, W. Barabash and C.R. Hill, A Type-checking Linkage System for PASCAL, Proceedings of 3rd International Conference on Software Engineering, 1978.

[4] UW-PASCAL Reference Manual, Madison Academic Computing Center, University of Wisconsin - Madison, 1977.

[5] R.J. LeBlanc, Extensions to PASCAL for Separate Compilation, SIGPLAN Notices 13:9, 1978.

[6] S.R. Bourne, A. Birrell and I. Walker, ALGOL 68C Reference Manual, Cambridge University, 1975.

[7] R.L. Schwartz, Parallel Compilation: a Design and its Application to Simula 67, Computer Languages, Vol. 3, 1978.

[8] J. Palme, Part Compilation in High Level Languages, Report No. FOA-P-C-8306-M3(E5), Swedish National Defense Research Institute, 1971.

[9] J.G. Mitchell, W. Mayberry and R. Sweet, Mesa Language Manual, CSL-78-1, Xerox Palo Alto Research Center, February, 1978.